Multi-Instrument Intercalibration (MIIC) Design

Design Dictionary

The following terms are used throughout this document:

Term	Definition
Calibration Coefficients	Calibration Coefficients estimate how the Target instrument correlates to the Source instrument. For example, using a linear fit, they are the y-intercept and slope in the equation “y=mx+b”.
Entity bean	Java components that can persist their state to database table(s)
Ephemeris	Data that predicts a satellite orbit and its location at any point in time.
Equal angle gridding	The algorithm that runs on the OPeNDAP server to convert instrument data from its native spatial storage structure to a regular grid format. Used so target and reference data can use the same underlying grid definition.
Filtering Conditions	Restrictions on which target/reference instrument data to use when computing calibration coefficients. For example “filter where the viewing zenith angle is less than 10 degrees”.
GEO	Geosynchronous Orbit satellite
IC Event	The Intercalibration Event is a moment in space/time when the target and reference instruments are observing the same physical area. This data is used to perform intercalibration.
IC Plan	One Intercalibration Plan represents an intercalibration workflow

	from start to finish. There can be many such plans for the same pair of reference & target instruments.
IFOV data	Instantaneous Field of View data. This is a geometric “footprint” tha most accurately describes where the instrument data was collected.
LEO	Low Earth Orbit satellite
Reference (Instrument/Data)	An instrument (and its associated data) used when performing calibration on a separate, target instrument.
REST interface	A service-oriented interprocess communication scheme. A REST interface is a set of URLs that can be called via HTTP requests to perform actions on a server.
RSR	Spectral Response Function. This is a table indicating relative spectral response values for different wavelengths.
Sampling Conditions	Restrictions on the algorithm that generates IC Events. For example, “search for IC Events during January 2011”.
Stateless bean	A component that performs processing for a client. “Stateless” means that all necessary information is provided via the interface o obtained from the database, never stored in the bean itself.
Target (Instrument/Data)	The instrument that we are calibrating.
TLE File	A Two Line Element file accurately states where a satellite was located at an instant in time. Typically these are generated daily by satellite tracking services and downloadable from the web.

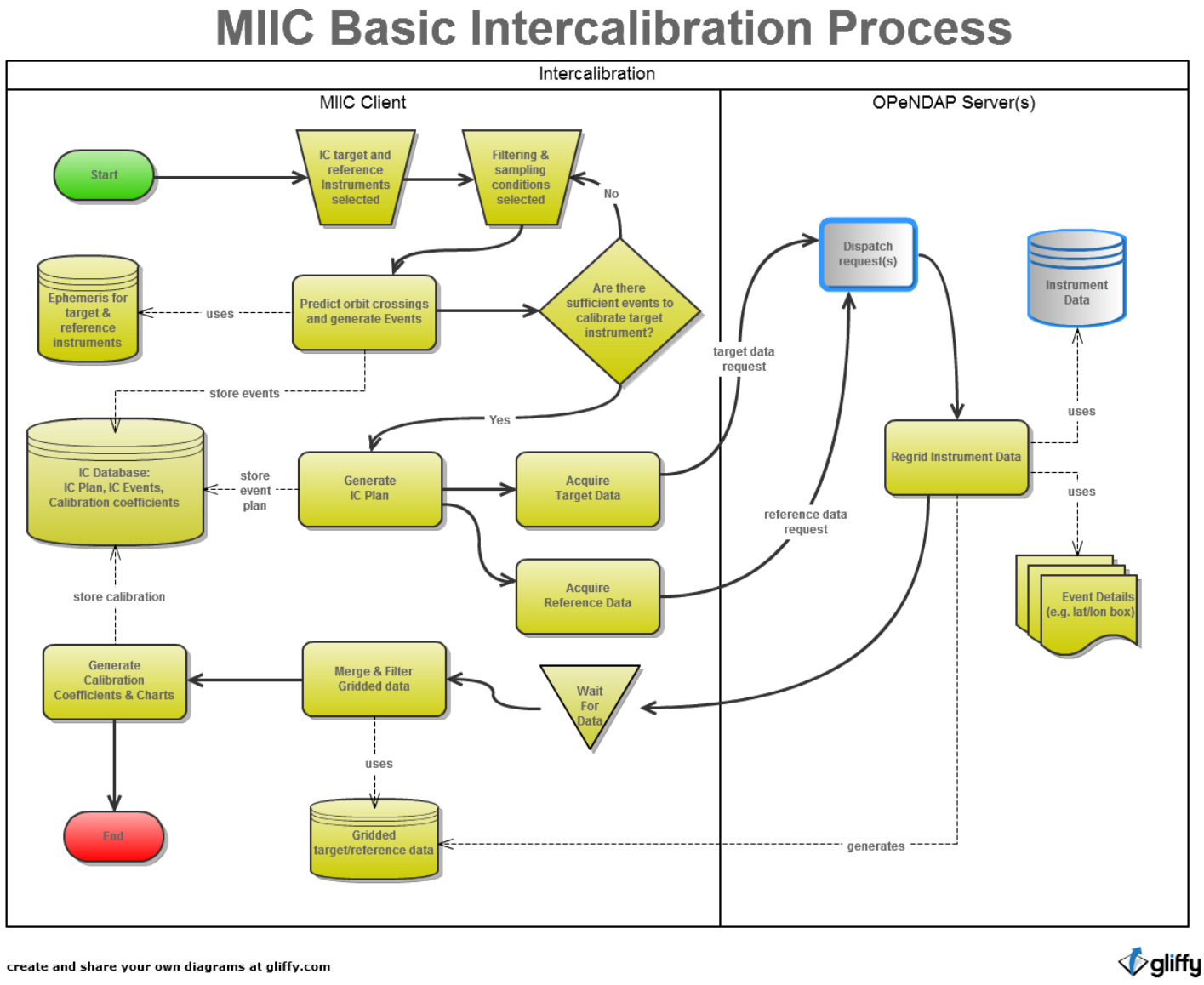
Introduction

This document describes the architecture and design of the Multi-Instrument Inter-Calibration (MIIC) Framework. Please refer to the document MIIC_Requirements for software requirements and use-cases.

The primary goal of MIIC is to support the satellite-to-satellite intercalibration process. Additional goals include data mining, data analysis, and supporting other large data intercomparisons. For example, comparing satellite sensor data with data predicted from earth science models.

Satellite Intercalibration Process

Performing an intercalibration involves finding where and when the target and reference satellites examine the same patch of earth, retrieving only that data of interest, and then performing a regression analysis to predict calibration coefficients.



The flowchart above describes the intercalibration (IC) process:

1. Select source and target instruments.
2. Select sampling and filtering conditions.
    a. Sampling conditions (e.g. a date range) will be used to determine which satellite orbit crossings to use.
    b. Filtering conditions restrict the instrument data used later during regression. They will be applied to the target and reference satellite data later in this process.
3. Generate IC Events. This requires obtaining TLE files for known satellite positions, predicting satellite locations at regular intervals, then finding where and when the orbit crossings are consistent with the sampling conditions. The IC Event consists of the start

and end times of the orbit crossing and its lat/lon extents. Start and end times (along with a-priori knowledge of satellite data products) are sufficient to determine which satellite data files are required to be downloaded.

4. Generate IC Plan. This is the set of IC Events we'll use for IC, along with all the criteria applied.
5. Acquire target/reference data. Download only the data of interest from an OPeNDAP server. The data acquisition requests differs based on what type of IC strategy we are performing.
    a. GEO/LEO IC data acquisition involves creating matching equal angle grids for the GEO and LEO data. LEO datafiles at resampled at the coarser resolution of a GEO instrument (we use a 1/2 degree Grid), given the lat/lon extents defined in the IC Event.
    b. LEO/LEO IC acquisition of IFOV data involves a two step process. First, the OPeNDAP server performs spectral convolution of the reference data according to the spectral response function of the target instrument. Second, the server performs spatial convolution of the target data according to the IFOVs returned from the reference data.
6. Merge and filter data. A single IC Event may contain multiple source and/or target data. These must be merged together. Unwanted data must be filtered out, according to the filtering conditions.
7. Generate calibration coefficients. Perform the regression analysis on the filtered data.
    a. Initially we are using a linear fit regression. This may be extended in the future to support other types of regression.

Data Analysis

The system also supports other data analysis features of interest to instrument science teams. These features include ad-hoc sampling of data from OPeNDAP servers and client-side data analysis.

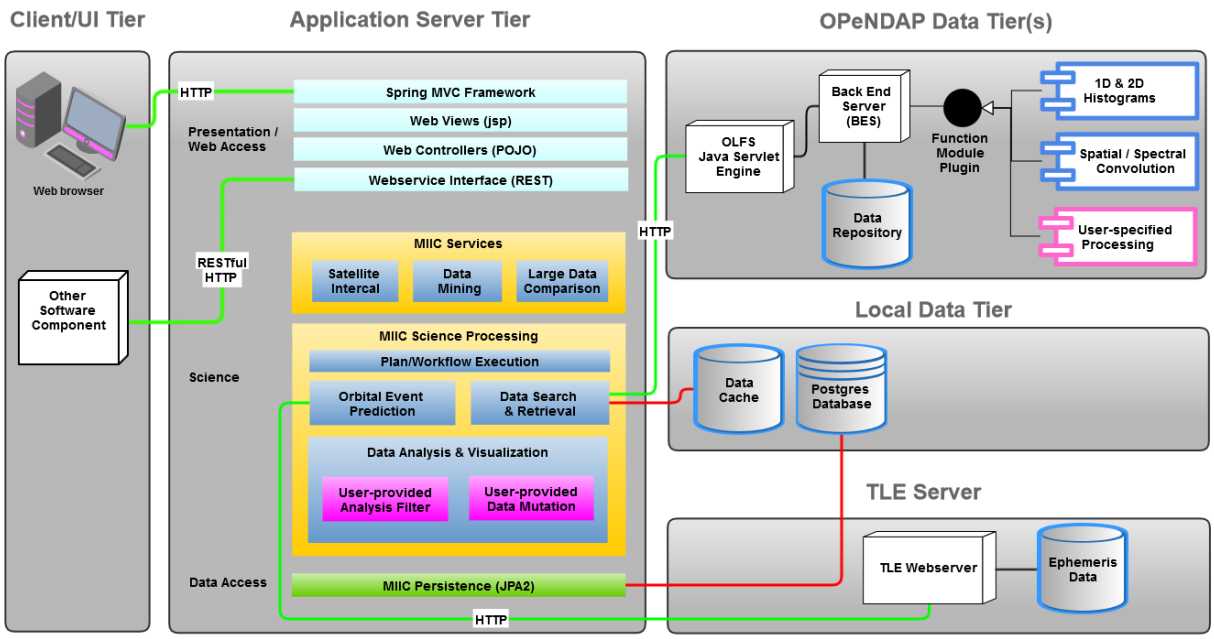
Given an instrument and date range, MIIC supports retrieving arbitrary 1D or 2D histograms containing frequency distributions and/or data averages from the server and merging them together.

Client-side data analysis features include generating charts showing raw or sampled instrument data, regression results, and histograms. Basic histogram operations are supported such as adding, subtracting, scaling, normalizing, and performing fits.

The client also supports trending calibration coefficients over time. (TBD - need more info here)

Multi-Tier Architecture

MIIC employs a Multi-Tier architecture with geographically distributed tiers where user interfaces, science-related processing, and intelligent data repositories are geographically distributed:



Client Tier

The Client/UI tier hosts the user interfaces for MIIC services. MIIC services include performing well-defined intercomparison workflows, and performing ad-hoc event prediction, data collection and data analysis.

The components hosted here can run on any workstation. We envision two potential types of MIIC clients:

1. Web pages used for managing the MIIC system and invoking well-defined use-cases.
2. Applications built by other groups that use MIIC as a service via its REST interface.

Application Server Tier

The Application Server Tier tier implements MIIC services. It is hosted on a Tomcat web server and provides common JavaEE capabilities:

- Allow multiple remote users to securely access and share system resources

(databases, etc)
- Allow compute-intensive tasks to run on a compute cluster for improved performance
- Provide MIIC services to external software clients
- Promote an open component-based design that can be extended into the future

Presentation Layer

The presentation layer generates web pages for users, data for web pages, and also provides a RESTful software interface.

REST Interface

The system provides a REST interface allowing software outside of the application tier to access MIIC services. REST was chosen over other webservice interfaces due to the simplicity of developing REST clients and servers.

The REST interface is defined as a set of URLs that can be called via a HTTP GET request to perform actions and return information. Calls that require additional data will use an HTTP POST or PUT call with an XML payload. All REST calls return information in an XML document.

For example, here is a URL to get an intercalibration plan bu its ID:
http://projects.mechdyne.com/miic/icplan/42

The request returns an XML document describing the desired IC Plan:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<icPlan>
  <state>DATA_ACQUIRED</state>
  <id>42</id>
  <tgt>
    <name>goes13</name>
    <satID>29155</satID>
    <type>GEO</type>
...
</icplan>
```

Science Layer

The Science Layer implements all MIIC services. It manages data intercomparisons and provides components that perform all science-related processing. The vast majority of code written for the MIIC project lives in this tier. Major components include:

- A plan executor that manages the processing steps of an intercalibration plan: predict

events, locate data, obtain data, perform analysis.
- An event predictor to find intercalibration events of interest
- A data locator to find needed data files by crawling OPeNDAP servers
- A data collector to generate the complex OPeNDAP server requests needed by the desired data collection strategy
- An analyzer to support arbitrary filtering and analysis of data obtained for the current plan

Data Access Layer

The data access layer supports finding, loading, saving and deleting objects in some sort of persistent store. The primary object to persist to the database is an ICPlan. For example the Data Access Object interface for an IC Plan is:

- `ICPlan loadPlan( int planID ); // load the requested plan ID`
- `List<ICPlan> findPlan( String name ); // find plans for the given user`
- `void savePlan( ICPlan ); // save or update the ICPlan in the database`

OPeNDAP Data Tier

The OPeNDAP Data Tier is accessed by components within the Application Tier to intelligently access data required by science processing. The OPeNDAP server is not simply a file server, it also has the ability to greatly reduce the amount of data transfer by pre-processing data files:

- Subrange the data whereby only data in a small geographic area are returned, or only data meeting some other filtering criteria
- Create a low-bandwidth representation of the data such as a 2D histogram or averaging of values
- Reinterpret the data in a meaningful way. For example convolve multi-spectral data to match the RSR of a desired target instrument.

OPeNDAP exposes URLs to process and acquire data files. OPeNDAP itself uses a 2-tier architecture. The OPeNDAP Lightweight Front end Server (OLFS), is a Tomcat hosted application to accept incoming HTTP requests and dispatch them to a Back End Server (BES). The BES server software handles reading data from data stores and filtering the data using server-side functions. The MIIC science tier communicates with the OLFS via REST calls.

The OPeNDAP server must also have a plugin that performs MIIC-specific data processing tasks. The plugin contains server side functions to generate histograms, spatial/spectral convolution, etc..

Design/Technology Rationale

A multi-tier architecture is a natural fit for a distributed system that has both back-end science processing and front-end client user interfaces. By hosting OPeNDAP servers on remote systems we gain access to huge volumes of earth science data wherever they happen to reside.

We further justify the need for a “middle” or application tier for these reasons:

- Despite relying on OPeNDAP servers to reduce the volume of data transfer we still anticipate the need for substantial data processing to perform required data analysis tasks
- We desire to provide a service that can be easily accessed by anyone at any time and from any device. Requiring them to deploy the service themselves would defeat this purpose.

Our chosen technology stack is PSTL: Postgres - Spring - Tomcat - Linux. Spring was chosen over other Java application servers (e.g. JBoss) for the following reasons:

- We have a small development team with experience using this technology stack
- Spring typically leads JavaEE developments. Spring was first to provide annotation-based configuration and dependency injection. Spring is currently at the forefront with new technologies like Grails (for rapid website development) and Spring Social (for integration with linkedin, facebook, twitter, etc.).
- Spring itself is lightweight in that you use only the frameworks or components that you need. Only a Tomcat container is required to deploy a Spring application.
- Spring integrates well with productive JavaEE frameworks like JPA2/Hibernate, JAXB, etc.
- Spring is open source and has a large community of users
- Spring dependency injection (DI) and Java annotations eliminate, for the most part, the need to use Spring or JavaEE interfaces in Java code.

Design Objectives

MIIC software will utilize service-oriented, layered, and component-based design. The service-oriented nature ensures that MIIC provides useful and reusable services to external clients. These external clients will not rely on MIIC software APIs or protocols but instead use simple HTTP-based requests along with our XML schema.

A layered design provides a clean separation between layers in the application server. Primarily we want to keep *application logic* (such as which web pages to present to users and what happens when users interact with UI components) separate from *science logic* such as how to talk to OPeNDAP servers and orchestrate an intercalibration workflow. Another major layering objective is to separate object persistence code from science logic.

Good component-based design is what allows the system to organically grow and accommodate new use cases. The general component design rules we follow are:

- Isolate or eliminate coupling to external tools, APIs and frameworks. This make their future replacement more plausible should the need arise.
- Create interfaces for all components. The interface describes the contract of the component but not its implementation. This supports clean and maintainable software.
- Prefer refactoring over exhaustive design. Due to the nature of rapidly evolving software it is unlikely a component design chosen in project infancy will suit the project its entire life. Therefore the goal should be to create a plausible design for the known requirements and allow it to evolve over time by refactoring.

Client Tier Technology Choices

MIIC web pages will use HTML5, jQuery UI, and CSS stylesheets. jQuery UI is a popular javascript framework for creating interactive and attractive user interfaces. It relieves the burden of integrating additional server-side component-based frameworks like JSF. It is also extendable and has the support of a large community of developers.

jQuery UI helps keep the web application responsive by rendering complex UI interfaces at the client and contacting the server for additional data only when necessary. Dynamic data requests to the server will be AJAX (asynchronous Javascript) requests returning JSON data (Javascript object notation).

The client will also integrate a Google Earth plugin to visualize geographic information at the client. The server will generate and return KML files that, for example, show geographical event regions using extruded polygons.

The client tier will communicate with MIIC via a RESTful interface. We require only proper use of the HTTP protocol, making our services as easy to consume as possible:

- POST methods are used to add new items to the server
- PUT methods are used to modify items at the server
- GET methods are used to retrieve items from the server, or to invoke actions
- Every HTTP Request must supply an Authorization Header with the client's credentials

The HTTP message body will normally be XML documents conforming to our schema. In some cases they may also be zip files containing raw data files.

Application Tier Technology Choices

Spring Model-View-Controller (MVC)

A presentation framework can generate web pages for users, and data for web pages or RESTful clients. Spring MVC was chosen as the presentation framework for these reasons:

1. Spring MVC simplifies integration with the Tomcat servlet container. Complicated servlet

configuration is limited to standard “boilerplate” items that are usually identical across projects. For example the servlet configuration has only one servlet and one filter to delegate everything to Spring.

2. Spring MVC enforces good UI design by separating the code to process incoming requests (the controllers) from the rendering of web pages (the views). The framework is further able to choose the type of view based on the client request. This enables data to be returned to the client as HTML, XML, or JSON as appropriate.
3. Spring MVC facilitates binding model data to web page form elements. This enables us to implement views using relatively simple JSP templates (see below). A Spring JSP tag library specifies how form elements like text boxes or lists bind to model data like the name of an intercalibration plan or a list of available variables from a dataset.
4. Spring MVC supports creating RESTful interfaces using the full HTTP protocol. For example, we can create a POST request to upload a new intercalibration plan or a PUT request to modify an existing plan.
5. Spring MVC does not force us to use any one specific technology for controllers or views, allowing these to change over time. Spring MVC is primarily focused on dispatching incoming requests to the right controller method, ferrying model data to the view, and invoking the appropriate view to return to the caller.

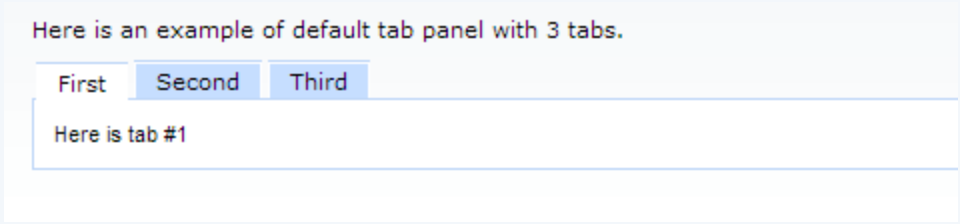
Java Server Pages (JSP)

JSP was chosen as the view technology due to its popularity and simplicity. We address the static nature of JSP pages and their relatively poor look-and-feel by using the jQuery UI javascript library. This provides interactive and attractive UI components without adding the server-side complexity of other frameworks.

For example, below is the JSF RichFaces code to render a set of tabs:

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    ... >
  <p>Here is an example of default tab panel with 3 tabs.</p>
  <rich:tabPanel>
    <rich:tab label="First">
      Here is tab #1
    </rich:tab>
    <rich:tab label="Second">
      Here is tab #2
    </rich:tab>
    <rich:tab label="Third">
      Here is tab #3
    </rich:tab>
  </rich:tabPanel>
</ui:composition>
```

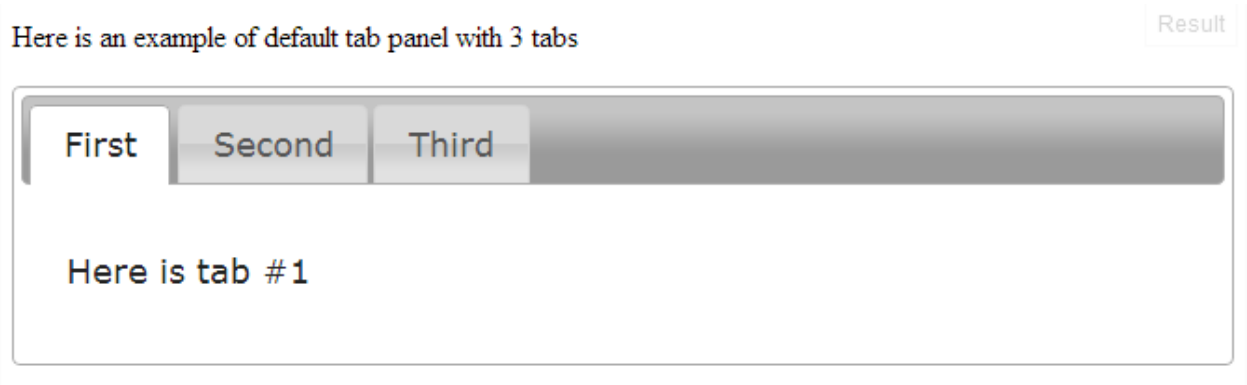
And it's default visual appearance on a web page:



Here is the corresponding version using jQuery UI and JSP:

```
<script>
 $(function() {
   $( "#tabs" ).tabs();
 });
</script>
<p>Here is an example of default tab panel with 3 tabs</p>
<div id="tabs">
 <ul>
   <li><a href="#tabs-1">First</a></li>
   <li><a href="#tabs-2">Second</a></li>
   <li><a href="#tabs-3">Third</a></li>
 </ul>
 <div id="tabs-1">
   <p>Here is tab #1</p>
 </div>
 <div id="tabs-2">
   <p>Here is tab #2</p>
 </div>
 <div id="tabs-3">
   <p>Here is tab #3/p>
 </div>
</div>
```

And it's default web page appearance:



Notice that the end result is similar but the JSP/jQuery UI version has clear advantages in terms of simplicity:

- It requires only basic HTML and javascript so can therefore run in any web browser
- It did not require integration of additional server-side UI frameworks

Spring Dependency Injection

One key challenge in complex systems is how to configure everything into a known and working state, given that:

1. The design itself is likely to evolve over time as new use cases are implemented
2. Our components require a lot of domain-specific information to work correctly

For example in order to download intercalibration data for an instrument you need to know:

- What the data product filenames look like
- How to compute data begin and end times for a file
- The variables contained in a file along with their valid ranges and dimensions
- Where and how to look for files on OPeNDAP servers
- How to correctly format the OPeNDAP requests to obtain data in the desired format

Hardcoding all this information inside Java code will create a brittle system and a maintenance nightmare. Our solution is to use the spring dependency injection (DI) framework to construct and configure most objects.

At its core, it is simply a way to “wire together” a bunch of objects with the state required to make a functioning system. An XML configuration file describes spring “beans”. A bean is usually a single instance of a Java class. Beans primarily provide arguments that are passed to the class’ constructor or setter methods.

Spring DI therefore offers the following benefits over not using a DI framework:

- Rapid configuration of complex object hierarchies. This saves time over writing custom configuration parsers and/or GUIs that save values to some other external store like a database table. Well-designed software typically has a large set of configuration values

that change frequently during development, and almost never change in production.
- Support for multiple configurations. For example we can create a test configuration to test a component in isolation. This can be much easier than, for example, standing up and populating servers with precisely the state needed for test.

In Spring DI, no coupling to Spring interfaces is required. The Spring component assembler uses only the constructors and setter methods of our Java classes.

For example, here is a spring bean to configure a “tleFactory” that knows how to generate two-line elements for satellites. It requires a property and a constructor argument. The property is a Data Access Object (DAO) for obtaining TLEs stored in our local database. The constructor argument is a TLE Provider to find TLEs not currently in our database. This configuration supplies a “CGITLE” which is a CGI-based web site that can return TLEs.

```
<bean id="tleFactory" class="gov.nasa.miic.tle.TLEFactory">
  <constructor-arg>
    <bean class="gov.nasa.miic.tle.CGITLE">
      <constructor-arg value="${TLEserverbase}"/>
    </bean>
  </constructor-arg>
  <property name="miicDAO" ref="miicDAO"/>
</bean>
```

Note that the URL “http://cloudsgate2.larc.nasa.gov” is provided via the property “TLEserverbase” through a separate Java property file which can be easily changed by a system administrator.

The fully configured “tleFactory” can now be supplied via spring config to other objects, for example the leo vs. geo event predictor implementation:

```
<bean id="leoGeoPredictor" class="gov.nasa.miic.eventprediction.PredictorImpl">
  <constructor-arg ref="tleFactory"/>
  <constructor-arg>
    <bean class="gov.nasa.miic.eventprediction.GEOvLEOswath"/>
  </constructor-arg>
  <property name="ephemerisTimestep" value="0.16666666666666666666666666666"/>
  <constructor-arg value="7"/>
</bean>
```

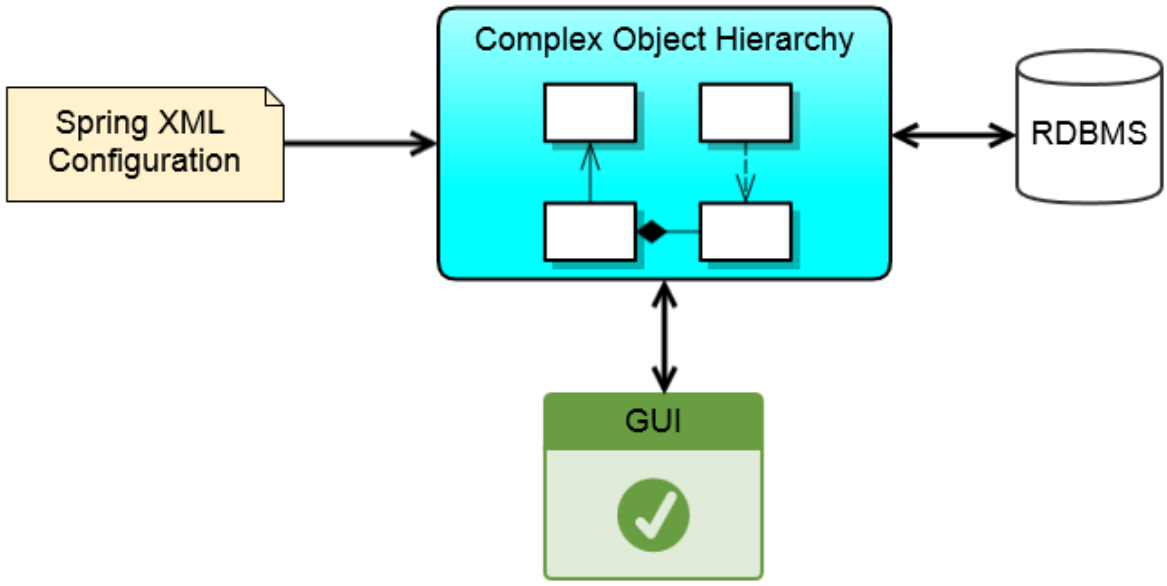
Spring DI Limitations

Due to their complexity, end users should never edit spring XML configuration files. A deployment where non-developers have to excessively edit spring configuration files indicates that either:

1. refactoring must be done to remove that burden, or
2. an administrative interface of some sort is missing.

Spring configuration files are also not intended for “round tripping” configuration information. Although Spring is very useful to configure complex objects, this should be viewed as strictly a one way “bootstrap” process. For example, the default configuration for a new deployment can be easily achieved via Spring XML configuration file. However if part of this configuration needs to change frequently we should store it to the database and possibly also develop a UI to facilitate changing it.

The following diagram illustrates these concepts:



JPA2/Hibernate

Hibernate is an object/relational mapping tool that can, with some effort, persist Java objects to database tables. Hibernate was originally developed as part of the JBoss technology suite, illustrating Spring’s ability to work with other Java frameworks. Since the introduction of Hibernate, the concept was found to be useful and was standardized using the Java Persistence Architecture (JPA).

JPA2/Hibernate was chosen as the persistence implementation for the following reasons:

- It supports flexible mapping of objects to/from database tables using Java annotation-based configuration.
- Spring provides wrapper classes that facilitate its integration.
- It is an industry standard and has benefitted from years of development and use
- It can automatically generate database schema, allowing the database to more easily evolve during development.
- We retain the ability to change databases in the future if that is required

Object/Relational mapping is however fairly complex so we mitigate this complexity by adhering to design rules:

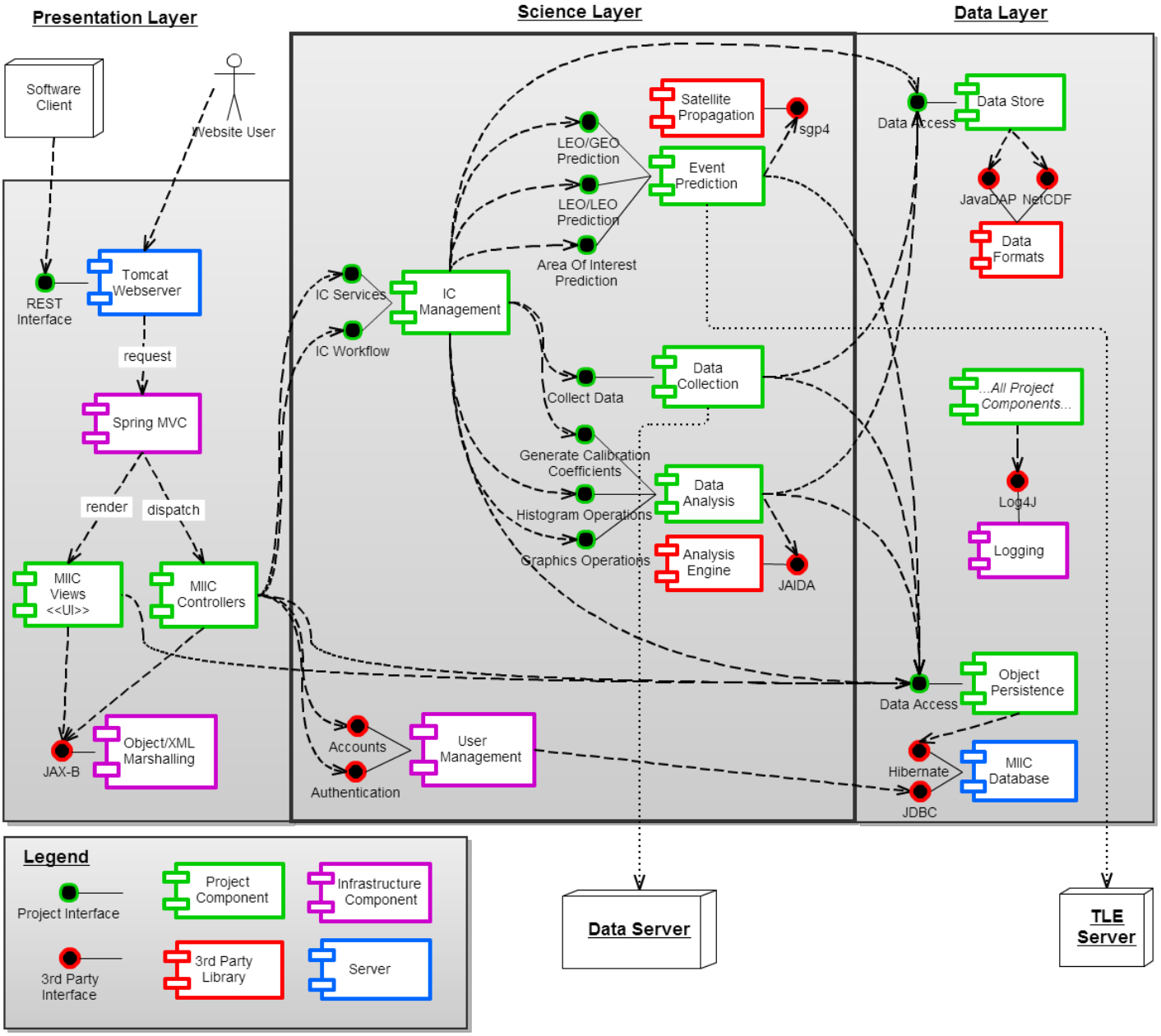
- Limit the number of classes using JPA persistence. For example we can delegate analysis implementation to a “Data Analyzer” and store that in a table as a binary large object, as opposed to making every possible Data Analyzer implementation a first-class database entity.
- Use JPA2 persistence annotations instead of XML configuration files. This makes persistence behavior much easier to understand in the code.
- Don’t take shortcuts when representing important relationships. For example if a satellite intercalibration plan references a data variable from an instrument, that relationship must be enforced by the persistence annotations, so it will be subsequently enforced by the database. This is trickier to get working correctly but will pay off in stability over the long run.

JAXB

The Java API for XML Binding (JAXB) transforms Java objects to/from XML documents. This enables us to support RESTful interfaces with very little effort. We use Java annotations to express how Java objects transform to XML elements and attributes where necessary.

Application Tier Architecture

This section describes the technical component architecture of the Application Tier. A good service-oriented component architecture allows us to ensure that the MIIC system exhibits loose coupling yet remains highly interoperable with other software.



The UML 1.x component diagram above shows the major components in the Application Tier (the color-coded boxes) and their external interfaces (the black circles). Connectors (the dashed lines) indicate components using interfaces provided by other components.

The components are color-coded:

- Green components consist primarily of Java classes (or other UI code) that must be re-used, refactored, or developed anew for this project.
- Purple components represent JavaEE frameworks that must be integrated into the system.

- Blue components represent external servers that must be configured for use by the system.
- Red components represent key 3rd party frameworks or libraries used by this project.

Presentation Layer Components

The Presentation Layer is responsible for handling all incoming REST and web page requests. All requests are first sent to the Spring MVC (model view controller) framework. Inside this framework, an object known as a *front controller* examines the request and dispatches it to the appropriate *controller* object in the MIIC Controllers component by examining the request type and/or its URL.

For web page requests a *controller* will typically load a *model* with application data using interfaces defined in the Science Layer. It will then invoke the appropriate *view* which will render the web page given inputs from the *model*. *Views* reside in the MIIC Views component.

For REST requests the typical *controller* action is to convert an XML payload to Java objects with the aid of the Object/XML Marshalling component. It may then invoke an interface on a component in the Science Layer, and convert the returned object(s) back into XML for the client to examine.

Science Layer Components

Components in the Science Layer implement MIIC services in a flexible way. The IC Management component is akin to a *front controller* for the science layer. It provides an interface for ic services such as Event Prediction, Data Acquisition, and Data Analysis. This interface is intended for clients that have a need for ad-hoc or exploratory large data comparisons.

The IC Management component also provides an ic workflow interface. The intention of this interface is to provide a simpler route for well-defined use cases, for example VIRS/GOES-13 intercalibration and trending of calibration coefficients.

In either case, the IC Management component will create and use an IC Plan to represent the state of the users' intercalibration tasks in the system. The IC Plan becomes the primary key in the database to access information such as intercalibration events, located data files, data obtained from servers, and analysis results.

The Event Prediction component is able to perform leo/leo predictions, leo/geo predictions, and leo/area of interest predictions. It uses the sgp4 library to generate satellite ephemeris and then computes intercalibration opportunities according to the requests' sampling conditions and other options.

The Data Collection component is able to locate and collect data from MIIC-compatible remote

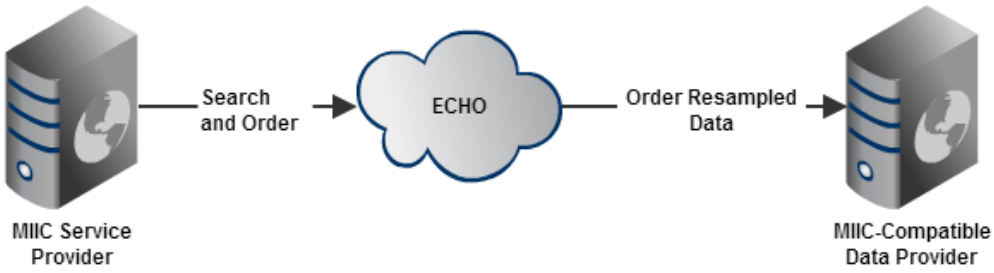
servers. It contains the intelligence to collect data according to the intercalibration strategy in use. The currently identified strategies are:

- Equal-angle gridding: resample the desired data over a regular grid (i.e. generate a histogram).
- Spectral convolution: reinterpret source data according to the spectral response function of the target sensor.
- Spatial convolution: resample target data according to the spatial definition of the source data

The primary data collector will locate data using a catalog search of well-known OPeNDAP servers that use the MIIC Plugin. It will then request data by using the OPeNDAP REST interface.

An alternate data collector may be developed to support the ECHO system for earth science data. This will broaden the appeal of MIIC by enabling access to more data for intercomparisons without having to enumerate servers a-priori, and without forcing data centers to use any specific technology.

In this scenario, MIIC will use the ECHO search and order interfaces to find and obtain the data. The data provider must also register an ECHO service for resampling data that is compatible with MIIC (see diagram below).



We can explore a potential ECHO integration without tackling the complexity of either designing and building new MIIC-compatible resampling services, or altering the MIIC client to accept intercomparison data in other formats. An easier approach would be to find and/or build an adapter that allows OPeNDAP servers to register as ECHO data providers and also as ECHO service providers for their MIIC-compatible resampling service.

Finally, the Data Analysis component provides interfaces to perform simple histogram operations, and graphics operations on data that has been retrieved. It can also generate calibration coefficients for sensors by using a histogram fit operation. This component uses the JAIDA framework to perform the data analysis. JAIDA supports a variety of fitting functions and methods, along with pluggable fitting engines like Minuit.

Data Layer Components

Components in the Data Layer provide access to persistent storage in databases and temporary files.

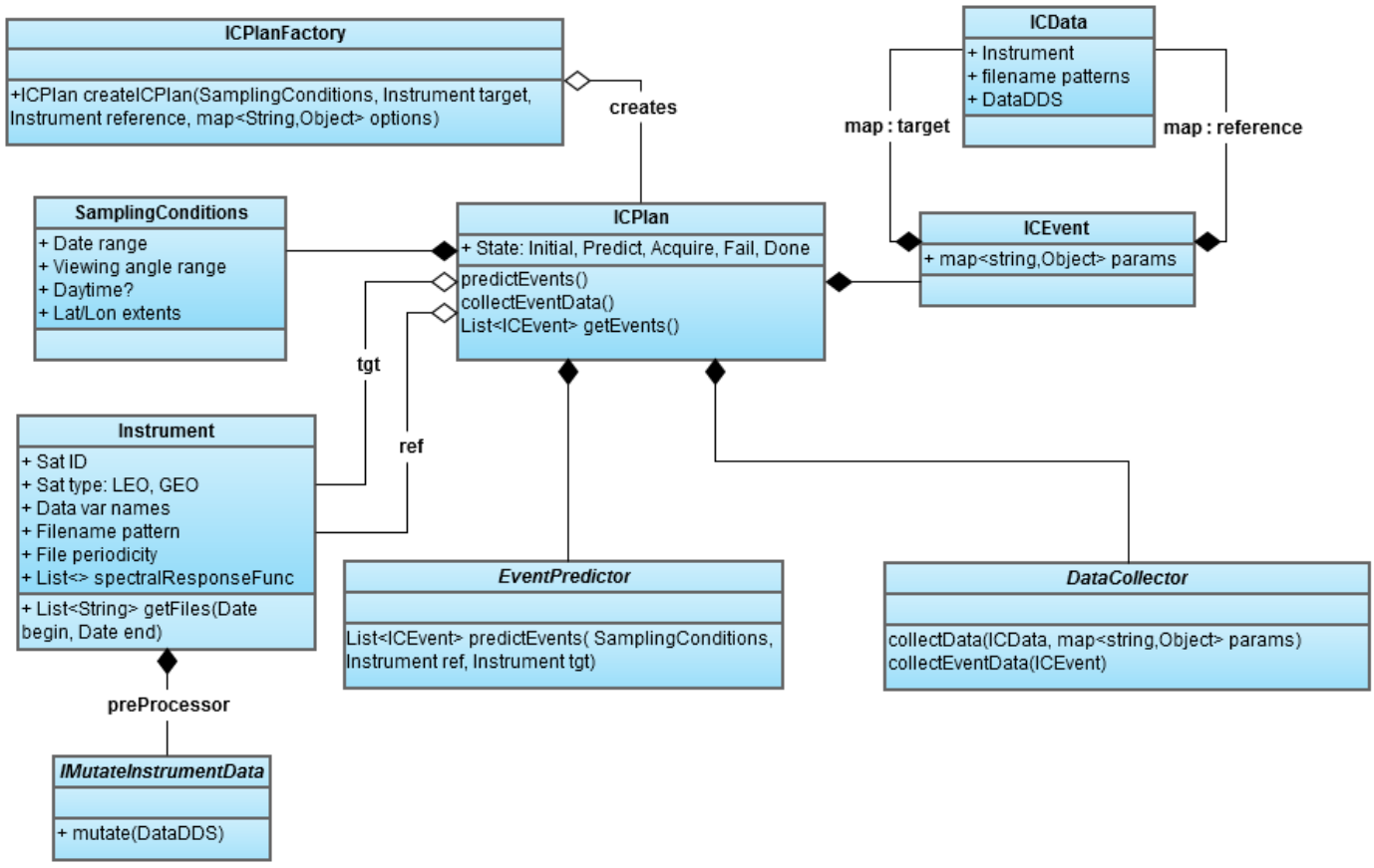
The Object Persistence component provides small *data access objects* that uses the Hibernate framework to perform create, read, update, delete and search operations on objects stored in a relational database. Examples of these objects include the IC Plan and generated Calibration Coefficients.

The Data Store component is able to temporarily store, retrieve, and access the raw data collected by the system and used during analysis. This component uses 3rd party Java libraries provided by OPeNDAP and Unidata to access data files stored in DAP and NetCDF formats. Any logic related to converting data into other formats will also reside in this component.

Application Tier Detailed Design

Satellite Intercalibration

The following UML diagram shows the core classes that support satellite-based Intercalibration:



Instrument

The instrument class stores all necessary information related to how we use satellite instrument data in the system. Of particular importance, an Instrument is able to determine what filenames store data for this instrument over a desired date range. Due to variances in how instrument files are ultimately named it actually generates regular expressions instead of specific filenames. We assume that there is one single authoritative source for instrument data and ignore issues like data reprocessing or multiple versions of the same file.

Because Instrument data files often don't include *all* the data we need, Instruments have an optional "preProcessor" derived from the IMutateInstrumentData interface. This can be used, for example, to generate viewing angles for GOES instrument data.

ICPlanFactory

The ICPlanFactory is used to generate new ICPlans suitable for a particular request. Clients supply SamplingConditions, which determine what data they are interested in, and the Target and Reference Instruments.

Clients may also provide options that can be used to override default behavior when executing the ICPlan. The specific settings available are:

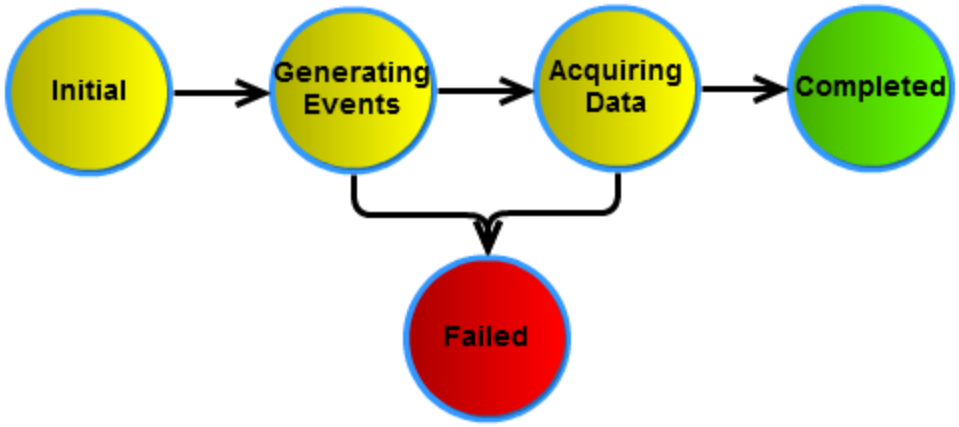
Option	Purpose
eagridsize	Determine the size of grid cells for equal angle gridding. Default is .5 degrees.
leogeocrosstracksamples	The number of samples in a LEO crosstrack to consider when checking for LEO instrument inclusion in the GEO code. Default 10.
TBD	TBD

The factory will examine the types of Instruments requested and construct an ICPlan that is best able to intercalibrate these instruments, or else throw an exception if intercalibration is currently not possible.

ICPlan, ICEvent and ICData

Each ICPlan describes a unique intercalibration session. Every intercalibration must be performed with exactly one Target and one Reference Instrument. ICPlans have SamplingConditions, which determine what data we are looking for. They also have a list of ICEvents, each defining a single instance where the instruments met the sampling conditions.

ICPlans have the following states:



The ICPlan does not reach the completed state unless all data for all ICEvents has been acquired. Note that the generation of calibration coefficients from ICPlans is handled elsewhere. This helps ensure that the data analysis features of MIIC are more general and not specific only to satellite intercalibrations.

The ICPlan will use parallelism when in the Acquiring Data state. It will use a ThreadPoolExecutor to ensure that data for multiple ICEvents can be fetched simultaneously.

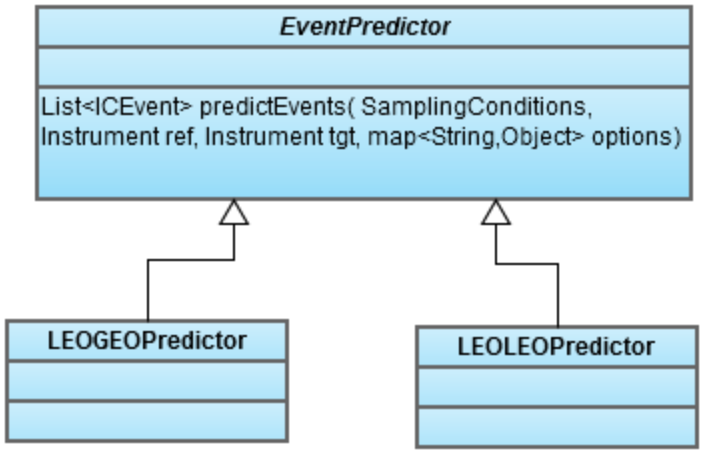
ICEvents store event parameters in a map along with the ICData that contains actual instrument data.The information is stored in a map to facilitate template generation code that creates the necessary OPeNDAP URLs to retrieve data.

ICData represents the reference and target instrument data from this ICEvent. Data can come from multiple files but will be merged together into a single OPeNDAP DataDDS structure.

EventPredictor and DataCollector

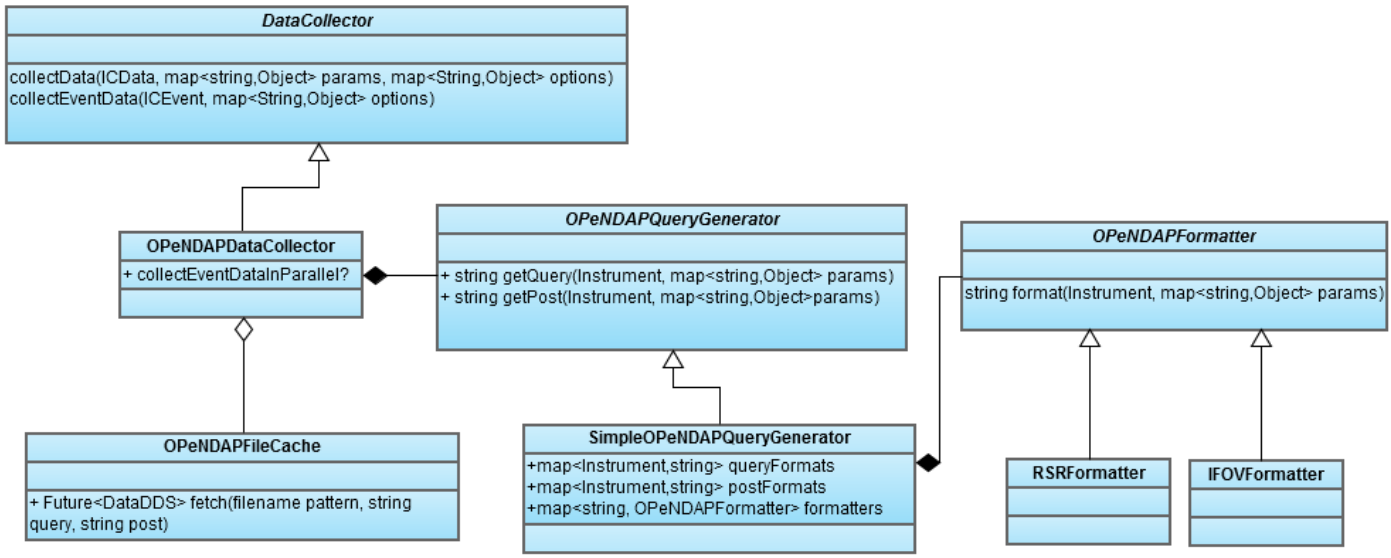
An ICPlan requires an EventPredictor, which is responsible for generating ICEvents, and a DataCollector, which is responsible for retrieving event data.

The following diagram shows the two event predictors available: one for LEO/GEO and one for LEO/LEO. These are adapted from Matlab code and their detailed design is out of scope for this document.



OPeNDAPDataCollector

The following diagram shows classes that collaborate to collect event data using OPeNDAP:



The OPeNDAPDataCollector is a concrete DataCollector for OPeNDAP servers that include MIIC extensions. OPeNDAP requests consist of a filename followed by a query string and POST data (TBD). The general strategy is to ask the OPeNDAPQueryGenerator to generate the URL components necessary to retrieve target and reference Instrument data for this event. It will then generate a new request for each filename and ask the OPeNDAPFileCache to fetch it. All data is then merged together and attached to the target or reference ICData as appropriate.

Note that the OPeNDAPDataCollector flag “collectEventDataInParallel” determines if we can request OPeNDAP data for the target and reference instruments in parallel. If this is set to false, we must first collect and merge together reference instrument data before collecting target instrument data. This is because some intercalibration strategies like IFOV spatial convolution require sending reference data footprints to the server when collecting target data.

The system uses parallelism when fetching ICEevent data from OPeNDAP. The OPeNDAPFileCache fetch method instantly returns a Future object that will eventually contain the DataDDS. In this way, we take advantage of parallelism offered by the OPeNDAPFileCache.

SimpleOPeNDAPQueryGenerator

The SimpleOPeNDAPQueryGenerator is a query generator that uses template substitution to generate OPeNDAP queries. For example, the template string for MODIS instruments using an equal angle grid strategy might be something like this:

```
?eamodisgrid({eagridsize},FILTER_VAR,Latitude,{latN},
{latS},FILTER_VAR,Longitude,{lonW},{lonE},...)
```

Where the strings in {} brackets are keys to access values from the ICEvents’ parameter map. The EventPredictor fills the map values for keys “latS”, “latN”, “lonE”, and “lonW”. In this example

the key “eagridsize” will have a default value of .5, which may be overridden by clients that specify an ICPlanFactory option.

RSRGenerator and IFOVGenerator

In the cases where simple string substitution with map values won’t cut it, there is an interface called *OPeNDAPFormatter*. This allows us to specify a more complex way to generate URL components.

The RSRFormatter class implements this interface and generates spectral response function data to send along with the request for spectral convolution:

```
?sciaspectralconv({targetRSR},...)
```

Where the key “targetRSR” points to an RSRFormatter object inside the SimpleOPeNDAPQueryGenerator. The RSRFormatter then generates a string containing the target Instruments’ RSR function data.

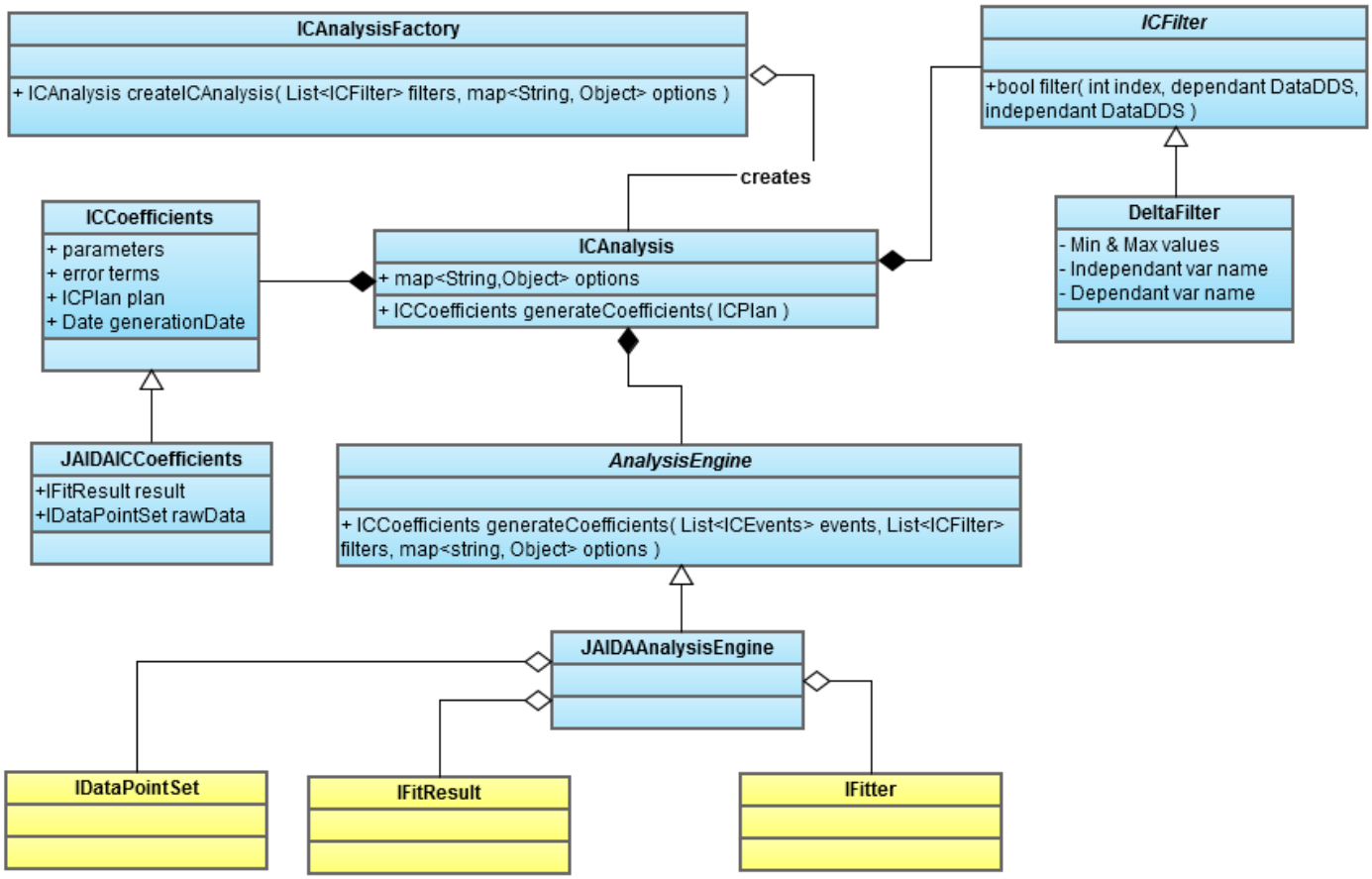
Similarly, the IFOVGenerator class generates IFOV footprints to send along with a call to spatial convolution:

```
?modisspatialconv({referenceIFOV},...)
```

Where the key “referenceIFOV” points to the IFOVGenerator object inside the SimpleOPeNDAPQueryGenerator. The IFOVGenerator generates a string containing the footprints from the reference data (each footprint is 10 floats: lat/lon of footprint corners and center).

Data Analysis Design

The following UML diagram shows the classes that generate calibration coefficients:



ICAnalysisFactory and ICFilters

The ICAnalysisFactory creates an ICAnalysis object that is able to generate calibration coefficients using the requested ICFilters and options. ICFilters determine what data to use during analysis in the case that ICPlan's data is still too noisy to obtain a good result. For example, the DeltaFilter can be used to filter where the difference in viewing angles is larger than 15 degrees.

The map of options specify how the analysis is to be performed. The following options are available:

Option	**Purpose**
fitterType	The type of fitting to perform, for example "Chi2" for chi-squared or "LS" for least squares.
fitterEngine	The desired fitting engine to use, for example "jminuit"
fitFunction	The type of function to which a fit is desired. For example, "p1" for a first degree polynomial.

dependantVarName	When multiple data variables are collected for the ICPlan, you must specify which variable to use during regression.
independantVarName	When multiple data variables are collected for the ICPlan, you must specify which variable to use during regression.

ICAnalysis and ICCoefficients

The ICAnalysis class uses an *AnalysisEngine* to generate ICCoefficients for an ICPlan. It is intended to be used over and over, maintaining a history of calibration coefficients over time.

Note that the analysis options and filters cannot be changed -- you must generate a new ICAnalysis instead. This is intentional so we can meaningfully compare calibration results.

JAIDAAnalysisEngine and JAIDAICCoefficients

The JAIDAAnalysisEngine performs analysis using the JAIDA data analysis library.

It must first process all the incoming ICEvent data, applying the ICFilters and creating an IDataPointSet for the dependant versus independent variable data.

Then, it performs the fit according to the analysis options. The analysis options map to the options available in JAIDA. The results of the fit is an IFitResult object.

Finally, it creates JAIDAICCoefficients to store the calibration parameters and error terms. This object also stores the IFitResult and IDataPointSet objects so that we can easily generate charts later.

Application Tier Interfaces

This section describes the external interfaces of the application tier. This includes how the system is configured, and its REST software interface.

System Configuration

A key aspect of the design is the use of spring XML “bean” configuration files. These files are used to “wire together” a working system as the software continues to evolve. For example, there is an engine required to predict orbit crossing events of two satellites. The interface is called *EventPredictor*. When intercalibrating GEO/LEO instruments we use the GEOvLEOswath implementation of this interface. The GEOvLEOswath class currently requires configuration settings for: dt, delta, deltac, sunscdeg. For flexibility, these values must not be hardcoded.

The corresponding section of configuration XML defining the bean might look like this:

```
<bean id="GEOLEOPredictor"
class="gov.nasa.miic.eventprediction.GEOvLEOswath">
  <constructor-arg><value>10</value></constructor-arg>
  <constructor-arg><value>50</value></constructor-arg>
  <constructor-arg><value>100</value></constructor-arg>
  <constructor-arg><value>90</value></constructor-arg>
</bean>
```

The “bean” GEOLEOPredictor becomes a singleton that can be used elsewhere in the configuration. Beans have constructor args and properties (i.e. getter/setter).

When configuring the factory that generates IC Plans, we indicate that modis/goes13 and ceres/goes13 intercalibrations should use the GEOLEOPredictor we defined earlier:

```
<bean id="ICPlanFactory"
class="gov.nasa.miic.common.ICPlanFactory">
  <property name="EventPredictorMap">
    <util:map>
      <entry key="modis,goes13" value="GEOLEOPredictor"/>
      <entry key="ceres,goes13" value="GEOLEOPredictor"/>
    </util:map>
  </property>
  ...
</bean>
```

Note that the XML above is only an example. The exact syntax depends on the class signature. For that reason, spring XML configuration files are generally not something that an end user would ever see or modify.

REST Interface

The REST interface allows software outside of the application tier to access MIIC functionality.

Request Type	**URL**	**Description**
GET	http://host:port/miic/rest/icplan	Returns a list of all ICPlan elements
GET	http://host:port/miic/rest/icplan/{plan_id}	Returns single ICPlan element
POST	http://host:port/miic/rest/icplan/create	Create a new ICPlan. Payload is ICPlan element.

GET	http://host:port/miic/rest/icplan/delete/{plan_id}	Delete an ICPlan and all of its associated data
POST	http://host:port/miic/rest/icplan/predict/{plan_id}	Generate new events for the plan, if not already done
GET	http://host:port/miic/rest/icplan/collect/{plan_id}	Retrieve data for all events, if not already done (OPeNDAP).
GET	http://host:port/miic/rest/icanalysis	Returns a list of all ICAnalysis elements
GET	http://host:port/miic/rest/icanalysis/{analysis_id}	Returns a single ICAnalysis element
GET	http://host:port/miic/rest/icanalysis/create	Create a new ICAnalysis. Payload is an icanalysis element.
GET	http://host:port/miic/rest/icanalysis/delete/{analysis_id}	Delete an ICAnalysis and all its ICCalibrations.
GET	http://host:port/miic/rest/icanalysis/calibrate/{analysis_id}/{plan_id}	Generate calibration coefficients for given plan. Returns an iccoefficient element.
GET	http://host:port/miic/rest/opendap	Retrieve list of all OPeNDAP data servers
GET	http://host:port/miic/rest/opendap/add	Add a new OPeNDAP server. Payload is opendapserver element.
	TBD	TBD

XML Schema Design

This section documents the XML format of system objects. XML is used to expose system data outside of the application tier. For example, a UI may use the “icevent” element to create a table showing the processing status of each IC event.

ICPlan

```
<icplan id=”33” target=”GEO” reference=”MODIS”
state=”completed” user=”aron”>
  <samplingcriteria from=”20120101” to=”20120102” latN=”33”
latS=”35” lonW=”120” lonE=”90” daytime=”true” vza=”15”
```

```
raz=”15” sza=”30”/>
  <description>MODIS/GEO IC Test for 6/2009</description>
  <eventpredictor id=”GEOLEOPredictor” began=”20130326:1217”
completed=”20130326:1220”/>
  <datacollector id=”OPeNDAPEAGRIDCollector”
began=”20130326:1220” completed=”20130326:1330”/>
  <icevents>
    <icevent id=”27” ...>
    ...
  </icevents>
</icplan>
```

The elements inside ICPlan represent its current state. Initially, it won’t have any ICEvents, because the predictor hasn’t run yet.

ICEvent

```
<icevent id=”27” state=”completed”>
  <properties>
    <prop name=”latN” val=”43”/>
    <prop name=”latS” val=”45”/>
    <prop name=”lonW” val=”110”/>
    <prop name=”lonE” val=”109”/>
    <prop name=”begin” val=”20120101:1440”/>
    <prop name=”end” val=“20120101:1502”/>
  </properties>
  <target instrument=”modis”>
    <filenames>
      <file
name=”MOD021KM.A2012001.1440.005.2012278082807.HDF”/>
      ...
    </filenames>

<url>http://clarreo-a/miic/rest/datadds/oke939kd309g</url>
  </target>
  <reference instrument=”goes13”>
    <filenames>
      <file
name=”MCIDAS.FY2E.2012.01.01.0501.08K.allch.bin”/>
      ...
    </filenames>

<url>http://clarreo-a/miic/rest/datadds/75yh459kd389</url>
```

```
  </reference>
</icevent>
```

The XML for icevents includes the event properties, the files retrieved, and URLs which can be used to access the raw OPeNDAP DataDDS for the event.

ICAnalysis

```
<icanalysis id="7">
  <icfilters>
    <icfilter type="DeltaFilter"
dependant="RelativeAzimuth_MEAN"
independant="RelativeAzimuth" maxDelta="15"/>
  </icfilters>
  <options>
    <option name="fitterType" value="LeastSquares"/>
    <option name="fitFunction" value="p0"/>
    <option name="dependantVarName"
value="EV_250_Aggr1km_RefSB_1_MEAN"/>
    <option name="independantVarName"
value="mcidas_data_0_MEAN"/>
  </options>
  <description>Linear fit with 15 degree rza</description>
  <analysisengine id="JAIDAAnalysisEngine"/>
  <iccoefficients>
    <iccoefficient icplanid="33" date="20130329:1344"
p0="33" p1="3.45" e0="0.001"/>
    ...
  </iccoefficients>
</icanalysis>
```

The iccoefficients inside the icanalysis represent all the analysis that has been performed to date for this icanalysis.

OPeNDAP Server-side Functions

Spatial/Spectral Convolution Functions

Equal Angle Grid Functions

Thee equal angle grid functions essentially return a 2D lat/lon histogram with averages of requested data variables. They also support filtering, both on the lat/lon extents, and restricting other data returned.

There are separate functions for equal angle gridding of MODIS, CERES, and GEO data files since the metadata is significantly different.

All equal angle grid functions return an OPeNDAP DataDDS structure:

```
Dataset {
    Structure {
        Float64 GRID_Latitude[#grid lat];
        Float64 GRID_Longitude[#grid lon];
        Float64 [Var name][_band name]_MEAN[Latitude = #grid
lat][Longitude = #grid lon];
        Float64 [Var name][_band name]_STD[Latitude = #grid
lat][Longitude = #grid lon];
        Int32 [Var name][_band name]_COUNT[Latitude = #grid
lat][Longitude = #grid lon];
    } Gridded_DATA;
} function_result_[filename];
```

The structure contains the following:

- Float arrays of GRID_Latitude and GRID_Longitude are 1D arrays of the Grid's lat and lon values.
- Three arrays for every requested data variable (and band chosen, if applicable)
    - Float array containing average data (size = #lat * #lon)
    - Float array containing standard deviation (size = #lat * #lon)
    - Int array containing cell counts (size = #lat * #lon)

eamodisgrid

The function signature is:

```
?eamodisgrid([cell size in degrees][,GRID_VAR,var name,[band
```

```
name,...]][,FILTER_VAR,var
name,min,max][,MIN_GRID_PTS,number min pts])
```

- GRID_VAR indicates variables in the file that should be gridded and returned, including "Latitude" and "Longitude". 3D variables require a band name and are expected to have size (#bands * #lat * #lon). 2D Variables are expected to have size (#lat * #lon).
    - Note that "TIME" is a special variable, generated from the file begin/end times. All data points for line N will have a time value of N/#lines times the total time of the MODIS file.
- FILTER_VAR indicates data variables that should be used to filter the gridded data. These must be 2D variables of size (#lat * #lon)
- MIN_GRID_PTS specifies the minimum number of data points that must reside in a grid cell to return data for that grid cell.

Here is an example to retrieve Lat, Lon, and EV_1KM_Emissive band 27 from -30 to 30 Lat and -100 to -80 Lon:
http://clarreo-a.larc.nasa.gov:8480/opendap/data/SampleData/MODIS/MOD021KM.A2010277.1710.005.2010278082807.hdf.ascii?eamodisgrid(.5,GRID_VAR,TIME,GRID_VAR,Latitude,GRID_VAR,Longitude,GRID_VAR,EV_1KM_Emissive,%2227%22,FILTER_VAR,Latitude,-30,30,FILTER_VAR,Longitude,-100,-80)

eageogrid

The function signature is nearly the same as eamodisgrid:

```
?eageogrid([cell size in degrees][,GRID_VAR,var name,[band
index,band missing value]...][,FILTER_VAR,var name,min,max]
[,MIN_GRID_PTS,number min pts])
```

- GRID_VAR indicates variables in the file that should be gridded and returned. The available variables are always "latitude", "longitude", "TIME", and "channel_data". Channel_data is a 3D variable and requires a band index and missing value.
    - Note that "TIME" is a special variable, generated from the file begin/end times. All data points for line N will have a time value of N/#lines times the total time of the GEO file.
- FILTER_VAR indicates data variables that should be used to filter the gridded data.
- MIN_GRID_PTS specifies the minimum number of data points that must reside in a grid cell to return data for that grid cell.

Here is an example to retrieve Time, Lat, Lon, and channel 0 data from -30 to 30 Lat and 100 to 180 Lon:

http://clarreo-a.larc.nasa.gov:8480/opendap/data/SampleData/GEO/MCIDAS.FY2E.2012.05.16.0501.08K.allch.bin.ascii?eageogrid(.5,GRID_VAR,TIME,GRID_VAR,latitude,GRID_VAR,longitude,GRID_VAR,channel_data,%220%22,-99,FILTER_VAR,latitude,-30,30,FILTER_VAR,longitude,100,180)

User Interaction Design